Formulating a novel barcode standard for the storage of information as colors

Is it possible to develop a standard for a hypothetical barcode which uses color to maximize information capacity while maintaining reliable operation?

Word Count: 3953
Candidate Number: 006702-0037
Computer Science Extended Essay

Table of Contents

Introduction **3**

Section A: Encoding **4**

Background 4

Standard Definition 5

Examples and Addendum 7

Section B: Header **8**

Background 8

Standard Definition 8

Examples and Addendum 10

Section C: Structure **11**

Background 11

Standard Definition 11

Examples and Addendum 13

Section D: Experimentation **15**

Background 15

Data Collection 16

Experiment 18

Conclusion **19**

Bibliography **20**

Introduction

In 1981, Internet Protocol version 4 (IPv4) was released. It was a standard for internetworking that allowed devices to exchange information, thus forming the internet. IPv4 gave each device a unique 32-bit address, restricting the possible number of devices with unique addresses to around 4.3 billion (Grinius, 2024).

As the internet grew, it became clear that 4.3 billion addresses were not enough. The internet was approaching its physical limit, and the standards we had in place failed to meet our demands for information. Facing IPv4 Address Exhaustion in 2017, a new standard capable of holding 3.4×10^{38} unique addresses, called IPv6, was developed (Deering et al., n.d.)

As humanity advances through the information age, the ways in which we communicate information and the standards that govern them must advance as well. Yet unlike IPv6, standards for 2D barcodes have remained relatively untouched since their invention in 1987 (Rizwan, 2023). This paper seeks to develop a novel standard for a hypothetical barcode which uses color to maximize information capacity while maintaining reliable operation.

This paper will address the stated research question in two parts. The first section of this paper will be dedicated to the development of a hypothetical barcode standard, dubbed *FCode*. Once the standard is fully described, the second section will experimentally determine the viability of the developed standard. The conclusions drawn from the experiment will determine whether the developed standard achieved the goals set in the research question.

Success in achieving the research question would prove optimistic for potential endeavors into colored barcode technology. Fields like pharmaceuticals, where drug information must be compacted onto tight bottles, supply chains, where information about a products processing journey must be condensed onto the product itself, and healthcare, where a patient needs quick access to extensive medical records, all stand to benefit from the successful implementation of this technology.

Section A: Encoding

Background

To begin developing the barcode standard, we must standardize how colors will be used to store information, or more technically, how the information is encoded.

For standard black/white 2d barcodes, the encoding process is straightforward. Since computers use base-2 (binary) to hold information, the smallest unit of information (a bit) has two states (either a zero or a one). Coincidentally, black and white codes also have two states (either black or white). As a result, the data processing can be trivialized by simply assigning each binary state to its own color.

The barcode described by this standard uses RGB colors to store more data per pixel. This complicates things, as one pixel will no longer correlate to exactly one bit. Since there are several ways to condense multiple bits into a single pixel, an encoding process must be standardized.

> RGB stands for [Red, Green, Blue], as it explains how an RGB pixel works. These pixels can create millions of colors by combining different amounts of red, green, and blue. Each pixel is described by the amount of red, green, and blue it has, with values for each ranging from 0 to 255.
>
> In array notation, a pure red pixel would be defined as [255,0,0], a pure green pixel would be [0,255,0], and a pure blue pixel would be [0,0,255]

The simplest encoding process would be a conversion from base-2 computer data to base-256, and then grouping the base-256 numbers into threes, each group representing a new pixel.

However, this is undesirable due to its lack of distinction between values. As opposed to a normal barcode, whose pixels can hold two distinct values (black or white), a single RGB pixel can hold 256 values *per color channel*, for a total of 256*256*256 (around 17 million) distinct values *per pixel.* While this sounds like a large amount of information, the difference between each value is so miniscule that no camera could ever feasibly read a barcode of this encoding.

As seen in *fig.1*, the difference between two codes may be indistinguishable, but if encoded using the prior process, could be drastically different in value. This poses an issue if the code were to be scanned by an imprecise camera, so there will have to be some “color distance” between one value and the next to allow room for error. A larger color distance would allow for more error but reduce the total storage capacity for a single pixel, whereas a smaller color distance could increase the storage space but decrease overall “scan-ability” for more imprecise cameras.

Figure 1. The difference in color between two RGB samples

This color distance shouldn’t be enforced by the standard, however, as it would make little sense to force an imprecise camera to use a slight color difference, or a precise camera to use a large color distance. Instead, the color distance a given code uses should be dependent on its intended application (commercial or industrial).

Additionally, it would be beneficial if the color distance could vary for certain color channels, as most cameras have better resolution for certain colors. Cameras with Bayer Filters, for instance, dedicate 25% of their photosensors to red, 25% to blue, and 50% to green. The green majority "approximates human photopic vision" (Bull, 2014), as our color-sensing cones are evolutionarily biased to green (due to chlorophyll's abundance in nature).

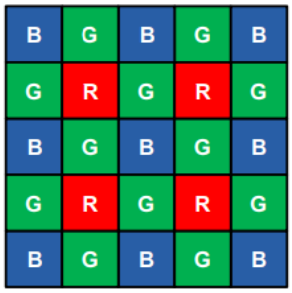


Figure 2. an example of a Bayer arrangement found in virtually all CMOS cameras (Bull, 2014)

Standard definition

To increase compatibility and address concerns noted in the background section, the encoding standard will be defined as such:

An FCode's encoding is described by its *channel order* and *codex* through a process dubbed *Zoingus encoding*. This encoding process is designed to maximize the total ways information can be translated to color while keeping the method to do so relatively simple; allowing generated codes to be optimized for each specific use case. The standard is as follows:

1) **The *channel order* is a list of three integers, ranging from 0 to 2, that describes the order each channel will be incremented in during its binary counting.**

Counting will begin with the color channel(s) whose value is 0 in the *channel order*. Once the channel's value exceeds 255, they will overflow into the color channel(s) corresponding to 1 in *channel order*, which will then overflow into the channel(s) corresponding to 2 in the *channel order*. Once the RGB pixel reaches its maximum value, it will overflow into another colored pixel and revert to [0,0,0].

**Note that channels with order -1 will be skipped, and channels with the same order will be incremented simultaneously.*

2) The *codex* is a list of three integers, ranging from 0 to 255, that describes the amount each given channel will be incremented by.

Color channels with order 0 will be incremented by the value set by the *codex's* first entry. Similarly, color channels with order 1 will be incremented by the second entry, and order 2 channels by the third entry *(this is because programming languages are zero-indexed).*

Zoingus Encoding Examples

Channel Order [2,0,1] Codex [230,222,1] Encoded Value 5	(base-10) 0 = [0,0,0] (base-10) 1 = [0,230,0] (base-10) 2 = [0,0,222] (base-10) 3 = [0,230,222] (base-10) 4 = [1,0,0] (base-10) 5 = [1,230,0] **Code = [1,230,0]**
Channel Order [-1,0,-1] Codex [42,71,70] Encoded Value 3	(base-10) 0 = [0,0,0] (base-10) 1 = [0,42,0] (base-10) 2 = [0,84,0] (base-10) 3 = [0,126,0] **Code = [0,126,0]**
Channel Order [0,1,0] Codex [120,4,13] Encoded Value 6	(base-10) 0 = [0,0,0] (base-10) 1 = [120,0,120] (base-10) 2 = [240,0,240] (base-10) 3 = [0,4,0] (base-10) 4 = [120,4,120] (base-10) 5 = [240,4,240] (base-10) 6 = [0,8,0] **Code = [0,8,0]**
Channel Order [0,0,0] Codex [255,255,255] Encoded Value 5	(base-10) 0 = [0,0,0] (base-10) 1 = [255,255,255] (base-10) 2 = [0,0,0],[255,255,255] (base-10) 3 = [255,255,255],[255,255,255] (base-10) 4 = [0,0,0],[0,0,0],[255,255,255] (base-10) 5 = [255,255,255],[0,0,0],[255,255,255] **Code = [255,255,255],[0,0,0],[255,255,255]**

Figure 3. A table containing several examples of zoingus encodings

Examples and Addendum

Below is an excerpt of python code capable of producing color sequences using zoingus encoding, as well as a barcode that has been encoded with text using said code.

```python
from math import ceil, log


def max_value(bases, channel_order):
    sum = 1
    for i in range(len(bases)):
        if i in channel_order:
            sum *= bases[i]
    return sum - 1


def color_seq(cval, ccodex, cchannel_order, max=256):
    val = int(cval)
    codex = list(ccodex)
    channel_order = list(cchannel_order)

    assert len(codex) == len(channel_order)
    codex_len = len(codex)

    colors = [0] * len(codex)
    bases = [ceil(max / num) for num in codex]
    high = max_value(bases, channel_order)
    pix_cnt = ceil(log(val) / log(high + 1)) - 1

    channel_order.extend(channel_order * pix_cnt)
    for i in range(len(channel_order)):
        if channel_order[i] >= 0:
            channel_order[i] += len(codex) * (i // len(codex))

    codex.extend(codex * pix_cnt)
    colors.extend(colors * pix_cnt)
    bases.extend(bases * pix_cnt)

    for n in range(len(colors)):
        if n in channel_order:
            for i in range(len(colors)):
                if channel_order[i] == n:
                    colors[i] = int(val % bases[n]) * codex[n]

            val //= bases[n]
        if val == 0:
            break

    return [colors[i:i + codex_len] for i in range(0, len(colors), codex_len)]
```

Figure 4. The code for generating zoingus encoded color sequences, as seen in the student's repository github.com/006702-0037/FCode/blob/main/encode.py

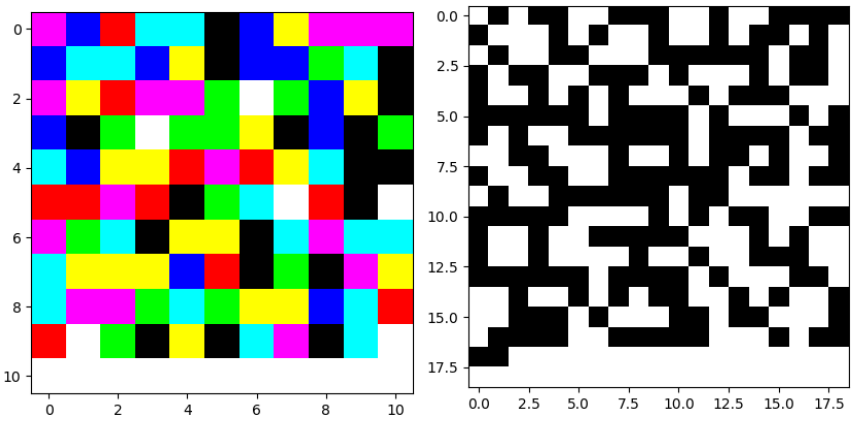


Figure 5. Left is a colored barcode representing the UTF-8 text in big endian, "apartment complex? i find it quite simple " with codex [255, 255, 255] and channel order [0,1,2], using 109 pixels. Right is its equivalent black and white barcode, which uses 327 pixels.

Section B: Header

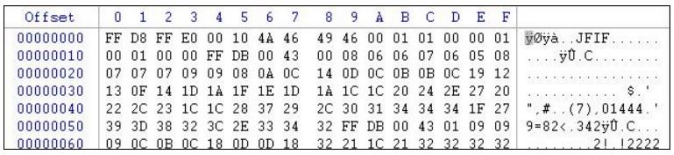
Background

Now that the encoding process is standardized, we must supplement encoded color data with information allowing it to be decoded.

Currently, barcodes generated in the FCode standard consist only of the colors output by a text's *zoingus* encoding. While theoretically this alone constitutes a color barcode, it lacks many features that a barcode needs to operate reliably. Given the example in Fig. 5, had the *channel order* or *codex* used in the barcode's encoding not been provided, it would be nearly impossible to decode.

Correcting this requires an element known as a *header*. A header is a chunk of data that contains "identifying information about the file and possibly metadata with information about the file contents" (Kent et al., 2006). Because this information is usually required before the computer can decode anything, it needs to come at the beginning (or, "*head*") of the file.

Since the header is simply binary data, it too must also be encoded. However, if we let the user decide what *codex* and *channel order* the header is in, we'd need a second header to define the encoding of the first one. To operate reliably, the standard must strictly define a header's possible encodings to facilitate readability, while still maximizing information capacity elsewhere.



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	FF	D8	FF	E0	00	10	4A	46	49	46	00	01	01	00	00	01	ÿØÿà..JFIF.....
00000010	00	01	00	00	FF	DB	00	43	00	08	06	06	07	06	05	08	....ÿÛ.C.....
00000020	07	07	07	09	09	08	0A	0C	14	0D	0C	0B	0B	0C	19	12	.....
00000030	13	0F	14	1D	1A	1F	1E	1D	1A	1C	1C	20	24	2E	27	20	..... $.'
00000040	22	2C	23	1C	1C	28	37	29	2C	30	31	34	34	34	1F	27	",#..(7),01444.'
00000050	39	3D	38	32	3C	2E	33	34	32	FF	DB	00	43	01	09	09	9=82<.342ÿÛ.C...
00000060	09	0C	0B	0C	18	0D	0D	18	32	21	1C	21	32	32	32	32	........2!.!2222

Figure 6. The header of a JFIF (also called JPEG) file as viewed through a hex editor. The signature "JFIF" text which appears at the beginning of every JFIF file can be seen on the right-hand side (Kent et al., 2006)

Standard definition

To maintain flexibility while enforcing a strict, definable guideline for headers, their standard will be as follows:

An FCode's header must be encoded using one of these two pre-set *zoingus* encodings:

1) ***High Level Header Encoding (HLHE)***
   Codex: [255,255,255]
   Channel Order: [0,1,2]
2) ***Low Level Header Encoding (LLHE)***
   Codex: [255,-1,-1]
   Channel Order: [0,0,0]

Furthermore, an FCode's header will be defined as exactly 40 bits at the start of a code, and the information those bits hold will be interpreted as follows:

1) Sanity Check: 4 Bits (# 1-4)
The Sanity Check bits ensure the header is decoded properly, as they should always read **1010**. By testing whether the Sanity Check bits are their correct values, systems can determine whether the header was decoded with the correct pre-set *zoingus* encoding.

2) **Endianness:** **1 Bit (# 5)**

Since different operating systems have different byte orders, bit #5 specifies the endianness (byte order) of the transmitted data. Its lookup table is as follows:

Little Endian	**0**
Big Endian	1

**Note that Bit Order is assumed to be Most Significant bit (MSB).*

3) **Data Type:** **3 Bits (# 6-8)**

The Data Type bits store information about what type of information is stored, giving the computer tasked with decoding an idea of how to interpret the data its reading. Its lookup table is as follows:

UTF-8 Data	**000**
ASCII Data	**001**
Binary Data	**010**
(Reserved for future compatibility in case of possible updates to standard)	**011,100,101,110,111**

4) **Codex:** **24 Bits (# 9-32)**

The Codex bits store the *codex* used to encode data appearing after the header. It consists of 24 bits, 8 bits per codex value. Codex values are padded with zeroes to fit the 8-bit size so that computers can distinguish between each codex value by length. Each 8-bit integer may be directly converted to its base-10 codex entry, so no lookup table is necessary. The first 8 bits correspond to the first value, the second 8 bits correspond to the second value, and the third 8 bits correspond to the third value.

5) **Channel Order:** **6 Bits (# 33-38)**

The Channel Order bits hold the *channel order* used to encode data appearing after the header. It consists of 6 bits total, 2 bits per *channel order* entry. The first two bits are the red *channel order*, the second two bits are the green *channel order*, and the last two bits are the blue *channel order*. The lookup table for each two-bit value is as follows:

Channel order 0	**00**
Channel order 1	**01**
Channel order 2	**10**
**Channel order -1 (unused channel)*	**11**

6) **Correction Level:** **2 Bits (# 39-40)**

The Correction Level bits indicate the degree of error correction codes that will be implemented by the Reed-Solomon error correction algorithm. Its lookup table is as follows:

No error correction codes	**00**
Low level error correction	**01**
Medium level error correction	**10**
High level error correction	**11**

**Note that this extended essay will not detail the standardization for error correction, as covering Reed-Solomon error correction would require an in-depth analysis of Galois theory, Linear algebra, and Finite field analysis, which is out of scope for this essay.*

Examples and Addendum

Below is an excerpt of python code capable of producing FCode headers, as well as an image visually depicting the structure of an FCode header.

```python
def num_to_bits(num):
    return "".join([x for x in '{:08b}'.format(num)])


def gen_heading(endian, dtype, codex, channel_order, correction_level):
    header = "1010"

    if endian == "big":
        header += "1"
    elif endian == "little":
        header += "0"
    else:
        return -1

    if dtype == "utf-8":
        header += "000"
    elif dtype == "ascii":
        header += "001"
    elif dtype == "bin":
        header += "010"
    else:
        header += "011"

    for i in codex:
        header += num_to_bits(i)
    for i in channel_order:
        header += num_to_bits(i)[6:]

    if correction_level == "none":
        header += "00"
    elif correction_level == "low":
        header += "01"
    elif correction_level == "medium":
        header += "10"
    elif correction_level == "high":
        header += "11"
    else:
        return -1

    if len(header) != 40:
        return -1
    return int(header, 2)
```

Figure 7. The code for generating an FCode header, as seen in the student's repository github.com/006702-0037/FCode/blob/main/heading.py

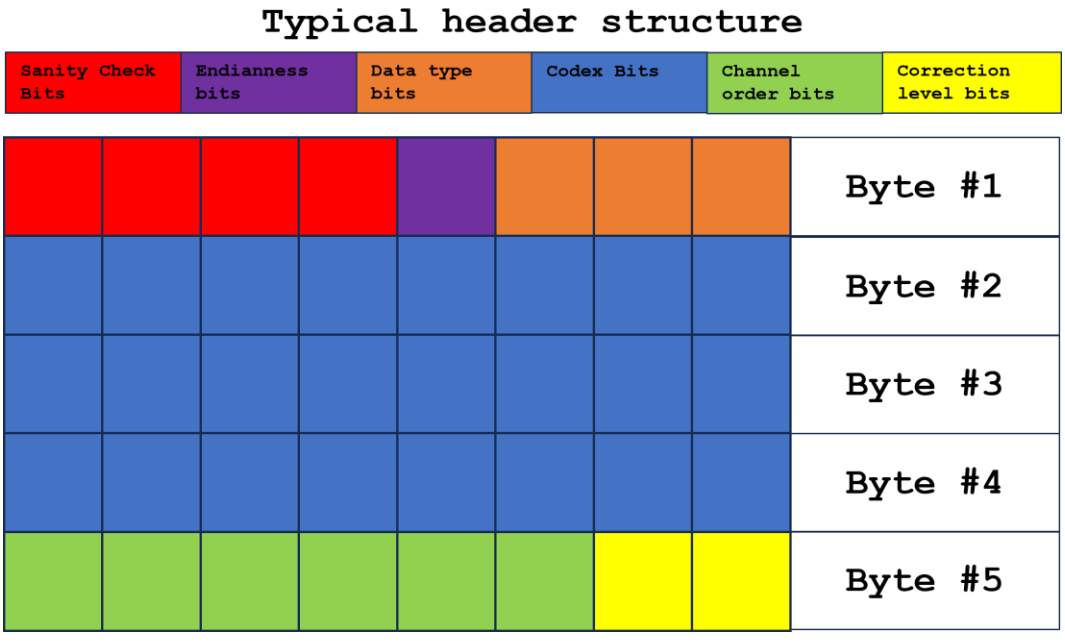


Figure 8. A diagram explaining the structure of a typical FCode header

Section C: Structure

Background

Now that information can be encoded and decoded in a reproducible manner, we must package it into a structure that safeguards it from potential errors.

Due to the addition of a header, it's now possible to decode an FCode where the recipient has no foreknowledge of its *codex/channel order*. This iteration would be suitable for digital file transfer, where corruption across devices is minimal if at all. However, for an FCode to be scannable in real-life applications, several more sources of error must be accounted for.

As seen in Fig. 9, these sources include (but are not limited to) rotation, reflection, axial-nonuniformity, grid non-uniformity, and modulation.

Some issues, such as print loss and color inconsistency, rely on proper printing of the FCode and cannot be enforced by the standard itself. Those which are addressable issues must be accounted for through the way color information is structured and what auxiliary features we add.

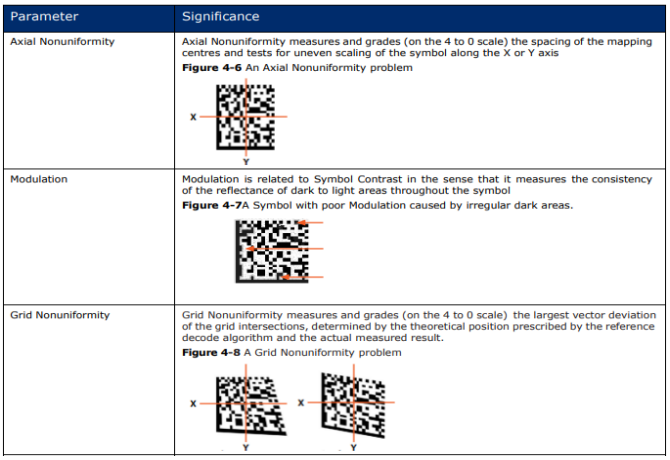

Parameter
Significance
Axial Nonuniformity
Axial Nonuniformity measures and grades (on the 4 to 0 scale) the spacing of the mapping centres and tests for uneven scaling of the symbol along the X or Y axis
Figure 4-6 An Axial Nonuniformity problem
Modulation
Modulation is related to Symbol Contrast in the sense that it measures the consistency of the reflectance of dark to light areas throughout the symbol
Figure 4-7A Symbol with poor Modulation caused by irregular dark areas.
Grid Nonuniformity
Grid Nonuniformity measures and grades (on the 4 to 0 scale) the largest vector deviation of the grid intersections, determined by the theoretical position prescribed by the reference decode algorithm and the actual measured result.
Figure 4-8 A Grid Nonuniformity problem

Figure 9. Excerpt from GS1 Datamatrix guideline, outlining examples of poor Datamatrix scans (gs1.org)

Standard definition

To address sources of error regarding real-life detection of FCodes, the standard regarding their design will be formulated as follows:

1) Barcode Data: (21.4% of total space)

The barcode data is the central aspect of the FCode layout, as it contains the actual data to be read by the computer. For this reason, it constitutes the largest section of the FCode's layout, and the remainder is designed to facilitate the reliable detection of this data.

2) Timing Pattern: (4x) (10.6% of total space)

The timing pattern is a series of alternating black stripes, each corresponding to the width of one pixel. These facilitate the reading of each row and column in the barcode data, since it provides the computer with a better understanding of the spacing (or 'timing') between each pixel.

3) **Color Correction Strips:** **(Shares space with Timing Pattern)**

The color correction strips help correct color inaccuracies produced by lighting, camera defects, or otherwise. They work by surrounding the color of indeterminate value (here, it's the barcode data) with a pattern of known color values (in our case it's the strips, with values [255,0,0], [0,255,0], and [0,0,255]). In doing so, a computer can correct any color offset by "applying a chromatic adaptation transform, from a calibrated color checker pattern included in the field of view." (Wannous et al., 2012) In general, this practice is known as colorimetry, and it's performed commercially with tools such as Pantone cards or Macbeth sheets.

4) **ArUco Markers:** **(3x) (5.3% of total space)**

ArUco markers are a type of "configurable square-based fiducial marker" (Garrido-Jurado et al., 2015) developed by the University of Cordoba and packaged in Python with OpenCV. Their most common application is pose estimation for augmented reality, as they can communicate information about their rotation, reflection, and localization within an image with a relatively small form factor. Leveraging open-source programs such as ArUco markers also serves to increase the adoptability of the FCode standard, since the code to detect them in images already exists and is widely implemented.

5) **Quiet Space:** **(15.0% of total space)**

The quiet space is a region of blank space surrounding the outside of a barcode which helps distinguish the barcode from its surroundings. It also prevents overlap from other barcodes when used in proximity and can reduce the impact of print defects (such as offset) by providing a margin of whitespace to accommodate misprints.

6) **Alignment Corner:** **(5.3% of total space)**

The alignment corner acts as a redundancy measure to ensure proper orientation of the barcode. The alignment corner is redundant, as the ArUco markers alone can transmit enough information for a computer to re-rotate them through mathematical transformations. Therefore, the alignment corner exists mainly to enforce orientation when being applied to surfaces and limits the amount of rotation correction required due to human error.

Examples and Addendum

Below is an excerpt of python code capable of generating formatted FCodes given a color sequence generated by the prior encode.py file.

```python
import numpy as np
import cv2

def create_aruco_marker(size, marker_id):
    dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_4X4_50)
    tag = np.zeros((size, size, 1), dtype="uint8")
    marker = cv2.aruco.drawMarker(dictionary, marker_id, size, tag, 1)

    return marker

def organize_color_seq(color_seq):
    square = int(len(color_seq) ** 0.5) + 1
    for _ in range(square ** 2 - len(color_seq)):
        color_seq.append([255, 255, 255])

    pixels = np.array(color_seq, dtype=np.uint8)
    pixels = pixels.reshape(-1, square, 3)
    return pixels

def GenerateFcode(color_sequence):
    pixels=organize_color_seq(color_sequence)

    # Define image dimensions
    width,height,colors = pixels.shape
    border_multiplier=1.15
    # Create the image with ArUco markers
    aruco_size = max([6*4,width, height]) // 4
    markers = [1, 2, 3]  # IDs for the three corners
    aruco_images = []

    for marker_id in markers:
        aruco_image = create_aruco_marker(aruco_size, marker_id)
        aruco_images.append(aruco_image)

    # Create a larger canvas for the resized image and markers with white borders
    canvas = np.ones((height + 2 * (aruco_size), width + 2 * (aruco_size), 3),dtype=np.uint8) * 255

    for i in range(aruco_size, width+aruco_size):
        canvas[:, i:i + 1] = [0,0,255]
    for i in range(aruco_size+1, width+aruco_size-1, 1 * 2):
        canvas[:, i:i + 1] = 0

    for j in range(aruco_size, height+aruco_size):
        canvas[j:j + 1, :] = [0,255,0]
        canvas[j, 0:aruco_size] = [255,0,0]
    for j in range(aruco_size+1, height+aruco_size-1, 1 * 2):
        canvas[j:j + 1, :] = 0

    canvas[aruco_size:-aruco_size, aruco_size:-aruco_size] = pixels

    # Place ArUco markers touching corners with the resized image
    canvas[:aruco_size, :aruco_size] = aruco_images[0]
    canvas[:aruco_size, -aruco_size:] = aruco_images[1]
    canvas[-aruco_size:, :aruco_size] = aruco_images[2]

    canvas = cv2.resize(canvas, (500,500), interpolation = cv2.INTER_AREA)
    height, width, channels = canvas.shape

    border_size=int(height*(border_multiplier**0.5))-height

    new_height = height+2*border_size
    new_width = width+2*border_size

    bordered_image = np.ones((new_height, new_width, channels), dtype=np.uint8) * 255
    bordered_image[border_size:-border_size, border_size:-border_size] = canvas

    # Display or save the resulting image
    cv2.imshow("Image with ArUco Markers", bordered_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Figure 10 The code for formatting an FCode color sequence into a barcode, as seen in the student's repository github.com/006702-0037/FCode/blob/main/structure_code.py

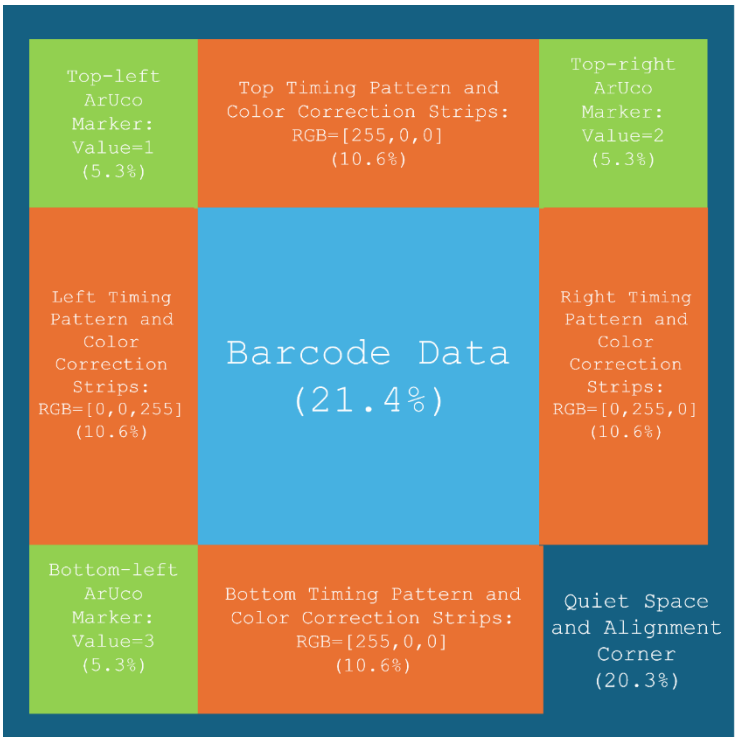

Top-left ArUco Marker: Value=1 (5.3%)
Top Timing Pattern and Color Correction Strips: RGB=[255,0,0] (10.6%)
Top-right ArUco Marker: Value=2 (5.3%)
Left Timing Pattern and Color Correction Strips: RGB=[0,0,255] (10.6%)
Barcode Data (21.4%)
Right Timing Pattern and Color Correction Strips: RGB=[0,255,0] (10.6%)
Bottom-left ArUco Marker: Value=3 (5.3%)
Bottom Timing Pattern and Color Correction Strips: RGB=[255,0,0] (10.6%)
Quiet Space and Alignment Corner (20.3%)

Figure 11 A diagram depicting the typical structure of an FCode barcode.

Figure 12 An example of a formatted FCode Barcode. This barcode contains the Big-Endian UTF-8 text, "Knock Knock!... Who's there?... Lettuce!... Thats impossible, lettuce cant speak.". This code was generated using channel order [0, 1, 2] and codex [255,255,255]

Section D: Experimentation

Background

To answer the research question of whether this standard "maximizes information capacity while maintaining reliable operation", we must experimentally determine the amount of information cameras can reliably distinguish.

The HCCB (High-Capacity Color Barcode) was a barcode released by Microsoft in 2007. It consisted of "rows of triangles of (up to eight) different colors, where consecutive rows are separated by a white line" (Querini et al., 2011). Much like the FCode standard, it stored data using colors to maximize its information capacity. Yet unlike the FCode standard, it was limited to a maximum of 8 colors. This is because the cameras of its time were unequipped to reliably detect much more, leading to its discontinuation.

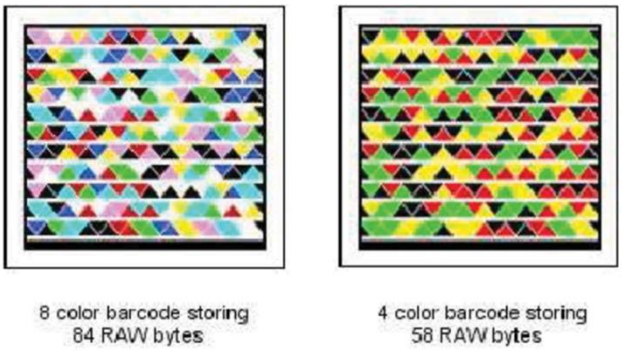


Figure 13 Two example HCCBs (Querini et al., 2011)

Advancements in consumer camera technology have exponentially increased camera quality during the 17 years since this product's release (Wiley), spurring a renewed viability for the use of colored barcodes. The student's hypothesis is that this increase in camera quality will allow for reliable storage of more information than previously comparable standards (*For the purposes of this experiment, "reliable" will be defined as operating with ≥80% accuracy, and the compared information storage value will be 8, as per the HCCB's 8-color storage*).

The experiment to test this will follow this general procedure:

- Collect data about various camera's color accuracies - quantified by some metric.
- Use this metric to calculate the optimal codex for each camera.
- Determine the average value each *codex* can maximally hold.
- Perform a noise analysis to determine the accuracy of the cameras.

If the above hypothesis is true, the average codex value should be >8, and cameras should be able to correct simulated noise from data encoded in their optimal codexes ≥80% of the time.

The most equivalent metric to that mentioned would be the Mean ΔE (MDE for short) of a camera. It's found by having a camera repeatedly scan a known color value (usually from a chart like a Macbeth grid) at a controlled illuminance (this experiment uses the D65 standard daylight illuminant) and then calculating the average distance the observed value is off by. However, the MDE does not measure the distance given two RGB values. It measures distance given two CIELAB values. This is important, as CIELAB is a perceptually uniform color space, whereas RGB is not.

A perceptually uniform color space "weights the same data variation equally all across the dataspace" (Crameri et al., 2020). This means that the numerical distance between colors is equal to the perceived distance between colors. To calculate the distance between two three-component color codes, you can use the Euclidean distance formula:

$$D = \sqrt{(A_2 - A_1)^2 + (B_2 - B_1)^2 + (C_2 - C_1)^2}$$

(Yuh, 2014)

In the formula above, D is the distance, and A, B, and C are the channels of the first and second color codes (for RGB, its [R,G,B] and for CIELAB, its [L*,a*,b*]).

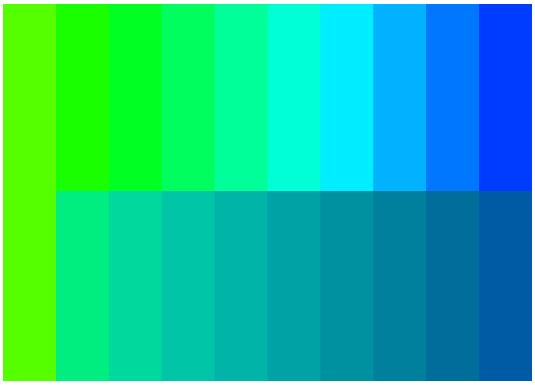


Figure 14 equal distance gradients as calculated using a non-perceptually uniform color space (top) and a perceptually uniform color space (bottom) (Madsen)

With this formula, it's possible to calculate a distribution of colors with equal distances between each other. However, as we see in Fig. 14, an even distribution as calculated in the RGB color space (top) is not actually uniform, whereas the distribution calculated in CIELAB (bottom) is, it is perceptually uniform while RGB is not.

Therefore, when we increase the gap between codex colors in the experiment, we must do so in CIELAB and convert back to RGB, otherwise some colors may be closer together than others, potentially increasing the amount of color inaccuracy.

Data Collection

Buying various high-end cameras to test this scientific hypothesis is expensive and unnecessary. Rather, we can collect data about these cameras and simulate the experiment virtually.

For this experiment, the student will utilize 88 different camera MDE measurements from Popular Photography's website, popphoto.com.

```python
from encode import max_value, color_seq
from math import ceil


# https://www.image-engineering.de/library/technotes/958-how-to-convert-between-srgb-and-ciexyz
def rgb_to_xyz(input_rgb):
    linear_rgb = [x / 3294.6 if x <= 10 else ((x / 255 + 0.055) / 1.055) ** 2.4 for x in input_rgb]

    X_D65 = 41.24564 * linear_rgb[0] + 35.75761 * linear_rgb[1] + 18.04375 * linear_rgb[2]
    Y_D65 = 21.26729 * linear_rgb[0] + 71.51522 * linear_rgb[1] + 7.21750 * linear_rgb[2]
    Z_D65 = 1.93339 * linear_rgb[0] + 11.91920 * linear_rgb[1] + 95.03041 * linear_rgb[2]
    return [X_D65, Y_D65, Z_D65]


def foo(x):
    return x ** (1 / 3) if x > 216 / 24389 else ((x * (841 / 108)) + 4 / 29)


def xyz_to_lab(input_xyz):
    L = 116 * (foo(input_xyz[1] / 100)) - 16
    a = 500 * (foo(input_xyz[0] / 95.0489) - foo(input_xyz[1] / 100))
    b = 200 * (foo(input_xyz[1] / 100) - foo(input_xyz[2] / 108.8840))
    return [L, a, b]


def rgb_to_lab(input_rgb):
    return xyz_to_lab(rgb_to_xyz(input_rgb))


# returns SQUARE of delta e, not actually delta e
def delta_e(lab1, lab2):
    return (lab2[0] - lab1[0]) ** 2 + (lab2[1] - lab1[1]) ** 2 + (lab2[2] - lab1[2]) ** 2


def get_best_codex_for_delta_e(de, order=(0, 1, 2)):
    codex = [1, 1, 1]
    delta_e_val = de ** 2

    best = 0
    best_codex = [-1, -1, -1]
    delta_avg = 0
    while True:
        bases = [ceil(256 / num) for num in codex]
        max_num = max_value(bases, order)
        delta_list = []

        bad = True
        if max_num >= best:
            bad = False
            colors = [[0, 0, 0]]

            for i in range(1, max_num + 1):
                current_rgb = color_seq(i, codex, order)[0]
                current_lab = rgb_to_lab(current_rgb)

                for c in colors:
                    delta_new = delta_e(c, current_lab)
                    delta_list.append(delta_new)

                    if delta_new < delta_e_val:
                        bad = True
                        break
                if not bad:
                    colors.append(current_lab)
                else:
                    break

        if not bad and len(delta_list) > 0:
            new_avg = sum(delta_list) / len(delta_list)

            if (max_num > best) or (max_num == best and new_avg > delta_avg):
                best = max_num
                best_codex = list(codex)
                delta_avg = new_avg

        if codex[0] == 255:
            codex[0] = 1
            if codex[1] == 255:
                codex[1] = 1
                if codex[2] == 255:
                    print(f"delta e: {de}, avg: {delta_avg}, val: {best}, codex: {best_codex}", flush=True)
                    return best_codex, best
                else:
                    codex[2] += 1
            else:
                codex[1] += 1
        else:
            codex[0] += 1

if __name__ == "__main__":
    from concurrent.futures import ProcessPoolExecutor

    import requests
    from bs4 import BeautifulSoup
    import re

    cams = []
    for i in range(1, 13):
        page_url = 'https://www.popphoto.com/search/camera+test/page/' + str(i)
        response = requests.get(page_url)

        soup = BeautifulSoup(response.text, 'html.parser')
        k = soup.find_all('a', class_='article-item__title')

        for i in k:

            camera_url = i['href']
            article = requests.get(camera_url).text
            match = re.search(r"average delta e.*?(\d+(\.\d+)?)", article, re.IGNORECASE)

            if match:
                if ',' in match.group(1):
                    if camera_url not in [i[0] for i in cams]:
                        cams.append((camera_url, float(match.group(1))))

    test_deltas = [i[1] for i in cams]
    print("finished webscraping for cams", flush=True)

    with ProcessPoolExecutor(32) as exe:
        # perform calculations
        results = exe.map(get_best_codex_for_delta_e, test_deltas)

    print("finished calculating optimal codexes", flush=True)
    with open('delta_e.csv', 'w', newline='') as csvfile:

        for i, r in enumerate(results):
            csvfile.write(",".join(
                [cams[i][0]] + r[1] + [cams[i][1]] + [str(cams[i][2]), str(r[2]), str(r[0][0]), str(r[0][1]),
                 str(r[0][2]) + "\n"]))
    print("finished writing csv. done.", flush=True)
```

Figure 15 The code used to collect data for the experiment. Due to the computationally expensive nature of these calculations, this code was run using the University of Florida's supercomputer, HiPerGator. This code is available at the students's repository github.com/006702-0037/FCode/blob/main/rgb_to_lab.py

URL	Delta E	Max Val	RGB Value
popphoto.com/story/gear-reviews/gear-2014-10-camera-test-leica-t	4.9	195	85,42,42
popphoto.com/camera-test-leica-q	6.3	143	85,49,48
popphoto.com/gear/2011/06/tested-leica-x1	7.7	107	107,48,51
popphoto.com/gear/2008/12/camera-test-fujifilm-finepix-1	7.6	107	101,49,51
popphoto.com/gear/2008/12/nikon-d90-camera-test	6.9	125	94,42,51
popphoto.com/camera-test-dxo-one-0	7.6	107	101,49,51
popphoto.com/story/gear-reviews/gear-2014-10-camera-test-leica-t-2	4.9	195	85,42,42
popphoto.com/gear/2008/12/camera-test-leica-m8	9.98	74	127,63,63
popphoto.com/gear/2009/07/camera-test-nikon-d5000	6.73	125	96,42,51
popphoto.com/gear/2013/10/camera-test-pentax-k-50	4.8	195	85,42,42
popphoto.com/camera-test-olympus-pen-f	8.7	89	106,51,63
popphoto.com/gear/2009/03/pentax-k2000-camera-test	8.3	89	111,51,63
popphoto.com/gear/2009/12/camera-test-sony-alpha-850	7.7	107	107,48,51
popphoto.com/gear/2012/12/sony-a99-camera-test	7.9	107	100,47,51
popphoto.com/gear/2012/01/camera-test-nikon-v1-ilc	7.4	107	103,50,51
popphoto.com/gear/2014/01/camera-test-pentax-k-3-dslr	7.2	119	85,52,46
popphoto.com/camera-test-panasonic-lumix-gx8	7.9	107	100,47,51
popphoto.com/gear/2014/07/camera-test-panasonic-lumix-gh4	7.8	107	100,48,51
popphoto.com/news/2009/06/olympus-e-620-camera-test	7.9	107	100,47,51
popphoto.com/gear/2008/12/camera-test-sony-alpha-350	6.9	125	94,42,51
popphoto.com/gear/2014/06/camera-test-nikon-d3300	6.4	143	85,48,48
popphoto.com/gear/2012/02/camera-test-pentax-q-ilc	6.5	125	100,42,51
popphoto.com/gear/2014/08/camera-test-nikon-d810	6.6	125	98,42,51
popphoto.com/gear/2014/06/camera-test-samsung-nx30	5.6	167	80,42,46
popphoto.com/gear/2014/11/camera-test-nikon-d750	6.7	125	97,42,51
popphoto.com/camera-test-sony-alpha-6300	7.8	107	100,48,51
popphoto.com/gear/2014/09/camera-test-samsung-nx-mini	7	125	93,42,50
popphoto.com/gear/2014/01/camera-test-nikon-d5300	6.8	125	95,42,51
popphoto.com/gear/2013/07/camera-test-samsung-nx300	5.5	167	81,42,47
popphoto.com/gear/2014/09/camera-test-pentax-645z	7.8	107	100,48,51
popphoto.com/gear/2013/03/camera-test-canon-eos-6d	7.3	119	85,52,46
popphoto.com/gear/2014/02/camera-test-fujifilm-x-e2	6.2	143	85,50,48
popphoto.com/gear/2011/05/camera-test-nikon-d5100-dslr	6.31	143	85,48,49
popphoto.com/gear/2008/12/camera-test-fujifilm-finepix-f30	7.94	107	100,47,51
popphoto.com/gear/2008/12/camera-test-olympus-sp-550uz	7.96	107	99,47,51
popphoto.com/gear/2008/12/sony-alpha-900-camera-test	9	89	104,51,61
popphoto.com/gear/2013/04/camera-test-nikon-d5200	6.9	125	94,42,51
popphoto.com/gear/2013/02/camera-test-sony-alpha-nex-6-ilc	7.9	107	100,47,51
popphoto.com/gear/2009/08/camera-test-canon-eos-50d	6.6	125	98,42,51
popphoto.com/gear/2008/12/camera-test-canon-eos-40d	7.7	107	107,48,51
popphoto.com/camera-test-canon-eos-5ds	7.3	119	85,52,46
popphoto.com/gear/2014/05/camera-test-fujifilm-x-t1	6.2	143	85,50,48
popphoto.com/gear/2013/06/camera-test-fujifilm-x100s	6	146	91,38,42
popphoto.com/gear/2008/12/camera-test-hp-photosmart-r927	9.82	74	127,63,63
popphoto.com/gear/2014/01/camera-test-leica-m-digital-rangefinder-camera	6.8	125	95,42,51
popphoto.com/gear/2013/10/camera-test-fujifilm-x-m1-camera	6.8	125	95,42,51
popphoto.com/gear/2008/12/camera-test-canon-eos-30d	7.61	107	100,49,51
popphoto.com/camera-test-nikon-d7200	6.7	125	97,42,51
popphoto.com/gear/2012/03/camera-test-samsung-nx200-ilc	7.3	119	85,52,46
popphoto.com/gear/2012/05/camera-test-nikon-d4-dslr	7	125	93,42,50
popphoto.com/gear/2013/11/camera-test-canon-eos-70d	7.3	119	85,52,46
popphoto.com/gear/2012/01/camera-test-nikon-j1-ilc	7	125	93,42,50
popphoto.com/gear/2014/06/camera-test-nikon-d4s	5.4	167	81,42,49
popphoto.com/gear/2008/12/camera-test-kodak-easyshare-v610	9.15	89	104,51,59
popphoto.com/gear/2010/05/camera-test-canon-eos-rebel-t2i	6.7	125	97,42,51
popphoto.com/camera-test/2011/09/ilc-test-littler-lumix	6.4	143	85,48,48
popphoto.com/gear/2013/06/camera-test-ricoh-gr-aps-c-compact	5.1	195	80,42,42
popphoto.com/camera-test-sony-a7r-ii	7.9	107	100,47,51
popphoto.com/gear/2008/12/camera-test-canon-powershot-sx100	9	89	104,51,61
popphoto.com/gear/2011/09/camera-test-sony-nex-c3-ilc	7.8	107	100,48,51
popphoto.com/gear/2008/12/camera-test-casio-exilim-ex-z600	6.89	125	94,42,51
popphoto.com/gear/2012/05/camera-test-fujifilm-x-pro1	9.3	74	127,63,63
popphoto.com/gear/2013/07/camera-test-canon-eos-rebel-t5i	6.7	125	97,42,51
popphoto.com/gear/2008/12/camera-test-mustek-mdc530z-digital-camera	13.39	49	133,53,63
popphoto.com/gear/2013/02/camera-test-fujifilm-x-e1-ilc	7.9	107	100,47,51
popphoto.com/gear/2009/08/camera-test-panasonic-lumix-dmc-g1	6.21	143	85,49,49
popphoto.com/gear/2013/09/camera-test-olympus-pen-e-p5	7.9	107	100,47,51
popphoto.com/gear/2008/12/camera-test-panasonic-lumix-dmc-tz1	8.37	89	110,51,63
popphoto.com/gear/2008/12/camera-test-panasonic-lumix-dmc-l1	9.49	74	127,63,63
popphoto.com/gear/2008/12/camera-test-canon-powershot-g9	8	107	99,47,51
popphoto.com/gear/2011/07/camera-test-panasonic-lumix-dmc-g3	6.1	143	85,51,48
popphoto.com/gear/2008/12/camera-test-sony-cyber-shot-dsc-h3	7.95	107	100,47,51
popphoto.com/gear/2013/11/camera-test-olympus-om-d-e-m1	7.9	107	100,47,51
popphoto.com/gear/2012/01/camera-test-panasonic-lumix-dmc-gx1-ilc	5.8	167	82,45,42
popphoto.com/gear/2012/10/camera-test-canon-eos-rebel-t4i	6.5	125	100,42,51
popphoto.com/gear/2012/04/camera-test-fujifilm-x10	7.4	107	103,50,51
popphoto.com/gear/2013/01/camera-test-nikon-d600-full-frame-dslr	6.8	125	95,42,51
popphoto.com/gear/2008/12/camera-test-fujifilm-finepix-f50fd	7.4	107	103,50,51
popphoto.com/gear/2013/04/camera-test-panasonic-lumix-dmc-gh3-interchangeable-lens-compact	6.5	125	100,42,51
popphoto.com/gear/2011/09/lab-test-sony-a77-new-king-aps-c-dslrs	7.2	119	85,52,46
popphoto.com/gear/2013/11/camera-test-sony-cyber-shot-rx100-ii-advanced-compact-camera	7.8	107	100,48,51
popphoto.com/camera-test-olympus-om-d-e-m5	7.9	107	100,47,51
popphoto.com/gear/2013/11/camera-test-sony-a7r-full-frame-interchangeable-lens-camera	7.9	107	100,47,51
popphoto.com/gear/2012/08/tested-sony-cyber-shot-dsc-rx100-compact-camera	6.6	125	98,42,51
popphoto.com/gear/2010/08/test-ricoh-gxr-a12-and-s10-camera-units	6.7	125	97,42,51
popphoto.com/gear/2008/12/test-samsung-nv10	8.22	89	112,51,63
popphoto.com/gear/2008/12/test-pentax-optio-a20	8.88	89	104,51,63
popphoto.com/gear/2011/04/lab-test-canon-t3i	6.7	125	97,42,51

Figure 16 The table of data collected by the program outlined in Figure 16

Experiment

To assess the hypothesis, the data collected in Figure 16 must now be tested in a camera noise simulation.

The collected data contains the camera's names, their stated MDEs, their optimal codex RGB values (calculated to keep the minimum distance between pixels around the camera's MDE), and the maximum values each codex can store.

The average maximum value is ~119, already exceeding the HCCB's maximum of 8, but proving the hypothesis means demonstrating *reliable* operation at this information capacity. That is, the cameras must average at least 80% accuracy when detecting data encoded with these codexes in a noise simulation.

To perform the simulation, each camera must repeatedly attempt to read a pixel's value given some degree of random color variation (the degree of color variation will be less than/equal to each camera's reported MDE, as this metric represents the average color offset expected by that camera). By dividing the number of codes read correctly by the total number of codes read, we can obtain the average accuracy across all cameras.

The simulation will be run for each camera's expected MDE (what each codex is optimized for) as well as under/overperforming scenarios, where camera MDEs are multiplied by a constant to increase/decrease their degree of color variation. In doing so, we can estimate the range in which an optimal codex can operate reliably outside its expected MDE.

Below is the code for the simulation and the graph it generated:

```python
import encode
from random import randint

def rgb_to_xyz(input_rgb):
    linear_rgb = [x / 3294.6 if x <= 10 else ((x / 255 + 0.055) / 1.055) ** 2.4 for x in input_rgb]

    X_D65 = 41.24564 * linear_rgb[0] + 35.75761 * linear_rgb[1] + 18.04375 * linear_rgb[2]
    Y_D65 = 21.26729 * linear_rgb[0] + 71.51522 * linear_rgb[1] + 7.21750 * linear_rgb[2]
    Z_D65 = 1.93339 * linear_rgb[0] + 11.91920 * linear_rgb[1] + 95.03041 * linear_rgb[2]
    return [X_D65, Y_D65, Z_D65]

def foo(x):
    return x ** (1 / 3) if x > 216 / 24389 else ((x * (841 / 108)) + 4 / 29)

def xyz_to_lab(input_xyz):
    L = 116 * (foo(input_xyz[1] / 100)) - 16
    a = 500 * (foo(input_xyz[0] / 95.0489) - foo(input_xyz[1] / 100))
    b = 200 * (foo(input_xyz[1] / 100) - foo(input_xyz[2] / 108.8840))
    return [L, a, b]

def rgb_to_lab(input_rgb):
    return xyz_to_lab(rgb_to_xyz(input_rgb))

def delta_e(lab1, lab2):
    return ((lab2[0] - lab1[0]) ** 2 + (lab2[1] - lab1[1]) ** 2 + (lab2[2] - lab1[2]) ** 2) ** 0.5


data_pairs = []
with open("delta_e_new.csv", 'r') as delta_file:
    delta_e_data = delta_file.read().splitlines()[1:]
    for i in delta_e_data:
        url, de, max_num, r, g, b = i.split(",")
        data_pairs.append((float(de), int(max_num), [int(r), int(g), int(b)]))


trials_each=1000
def simulate(modifier):
    current = []
    global trials_each
    for k in range(trials_each):

        test_data = data_pairs[randint(0, len(data_pairs) - 1)]
        color_1 = encode.color_seq(randint(1, test_data[1] + 1), test_data[2], (0, 1, 2))[0]
        noise_color = [0, 0, 0]

        while delta_e(rgb_to_lab(color_1), rgb_to_lab(noise_color)) > test_data[0] * modifier:
            noise_color = [randint(0, 255), randint(0, 255), randint(0, 255)]

        best_color = []
        best_delta = 1000
        for i in range(1, test_data[1] + 1):
            code_color = encode.color_seq(i, test_data[2], (0, 1, 2))[0]
            if delta_e(rgb_to_lab(code_color), rgb_to_lab(noise_color)) < best_delta:
                best_delta = delta_e(rgb_to_lab(code_color), rgb_to_lab(noise_color))
                best_color = list(code_color)

        if best_color == color_1:
            current.append(1)
        else:
            current.append(0)

    return sum(current) / len(current)

if __name__ == "__main__":
    from concurrent.futures import ProcessPoolExecutor
    import matplotlib.pyplot as plt

    trials_each = 1000
    runs = [0.5 + 0.1 * x for x in range(25)]
    with ProcessPoolExecutor(32) as exe:
        results = exe.map(simulate, runs)
    plt.scatter(runs,list(results))
    plt.show()
```

Figure 17 The python code used to run the experiment using the data collected. The code is available at the student's repository github.com/006702-0037/FCode/blob/main/test.py

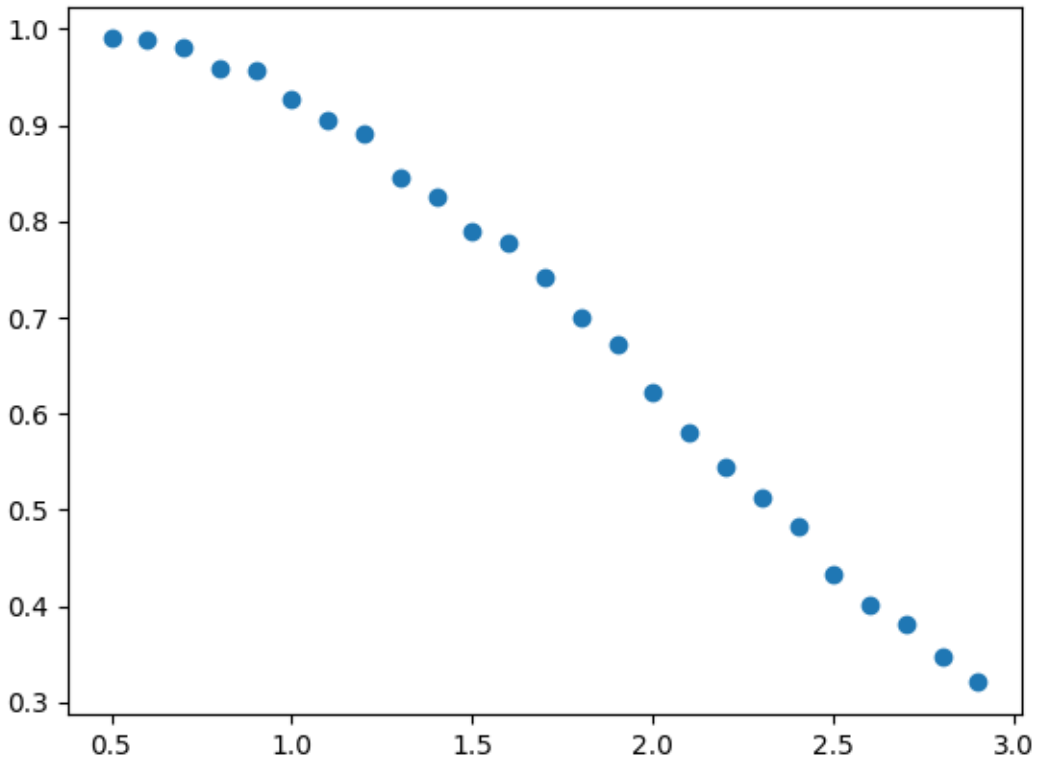


Figure 18 A graph of the average accuracy for each camera in the noise simulation (x1/100 scale, y-axis), vs the modifier applied to each camera's MDE (x-axis)

Conclusion

The graph above shows that, given a camera operating at its stated MDE (x=1.0), on average, it will read distorted values with ~90% accuracy. This value satisfies the conditions for "reliable operation" as defined under the background subheading of Section D.

Additionally, the graph shows that on average, accuracy remains reliable (≥80%) for cameras operating within a 50% difference of their stated MDE ($0.5 \leq x \leq 1.5$)

Because the barcode standard succeeded at having reliable operation (~90% accurate) and increased information capacity from previous standards (8→119 values), the hypothesis for this experiment, and therefore the research question, is correct; it is possible to develop a standard for a hypothetical barcode which uses color to maximize information capacity while maintaining reliable operation.

If the student were to replicate these results in the future, the principal change would likely be to conduct the experiment in real life using actual camera equipment, rather than simulated trials.

Bibliography

Bull, David R. “Digital Picture Formats and Representations.” *Communicating Pictures*, Academic Press, 25 July 2014, doi.org/10.1016/B978-0-12-405906-1.00004-0.

Gold, Aaron, et al. “Techniques and How-Tos, Camera Reviews.” *Popular Photography*, www.popphoto.com. Accessed 10 Mar. 2024.

Kent, Karen, et al. “Guide to Integrating Forensic Techniques into Incident Response.” *CSRC*, 1 Sept. 2006, csrc.nist.gov/publications/detail/sp/800-86/final.

Yuh, Adeline W. “Comparison of Color Accuracy of Commercial Digital Cameras and Cellular Phonecameras for Laboratory Purposes.” *The Research Repository @ WVU*, doi.org/10.33915/etd.179. Accessed 10 Mar. 2024.

Crameri, Fabio, et al. “The Misuse of Colour in Science Communication.” *Nature News*, Nature Publishing Group, 28 Oct. 2020, doi.org/10.1038/s41467-020-19160-7.

Querini, Marco et al. “2D Color Barcodes for Mobile Phones.” *Int. J. Comput. Sci. Appl. 8* (2011): 136-155.

Garrido-Jurado, S., et al. “Generation of Fiducial Marker Dictionaries Using Mixed Integer Linear Programming.” *Science Direct,* Pergamon, 1 Oct. 2015, doi.org/10.1016/j.patcog.2015.09.023.

“GS1 DataMatrix Guideline.” *GS1*, www.gs1.org/docs/barcodes/GS1_DataMatrix_Guideline.pdf. Accessed 10 Mar. 2024.

Madsen, Rune. “Perceptually Uniform Color Spaces.” *Perceptually Uniform Color Spaces - Programming Design Systems*, programmingdesignsystems.com/color/perceptually-uniform-color-spaces/. Accessed 10 Mar. 2024.

Wiley, Keith. “Digital Photography.” *Mind Ramblings: Digital Photography*, keithwiley.com/mindRamblings/digitalCameras.shtml. Accessed 10 Mar. 2024.

Wannous, Hazem, et al. “Improving Color Correction across Camera and Illumination Changes by Contextual Sample Selection.” *SPIE Digital Library*, SPIE, 25 June 2012, doi.org/10.1117/1.JEI.21.2.023015.

Grinius, Vincentas. “What Is Ipv4 Address Exhaustion and How Can We Alleviate It? - IPXO Blog.” *IPXO*, 15 Jan. 2024, www.ipxo.com/blog/how-can-we-alleviate-ipv4-exhaustion/.

Rizwan, Zara. “QR Code History: Evolution of the Popular 2D Barcode.” *Scanova Blog*, 4 Oct. 2023, scanova.io/blog/qr-code-history/.

“How to Convert between sRGB and CIEXYZ.” *Image Engineering*, 8 Aug. 2017, www.image-engineering.de/library/technotes/958-how-to-convert-between-srgb-and-ciexyz. (In code)

Deering, Dr. Steve E., and Bob Hinden. “RFC 8200: Internet Protocol, Version 6 (Ipv6) Specification.” *IETF Datatracker*, datatracker.ietf.org/doc/html/rfc8200. Accessed 10 Mar. 2024.